Operating Systems - Assignment

COMP2006

Jordan Yeo
17727626

Curtin University
Science and Engineering

Contents

Introduction

The following report is for the Operating Systems Assignment for 2017. It will detail how mutual exclusion was achieved for processes and threads. Also the processes and threads that accessed shared resources. Submitted alongside this report is a README, source code and test inputs and outputs.

1 Process MSSV

For this implementation of the solution, mutual exclusion is achieved by having the parent (consumer) wait for all child (producer) processes to complete execution before continuing. This waiting is implemented by having the parent wait for the semParent semaphore before accessing the value of the shared variable representing the number of child processes finished. Once a child completes execution is waits for the semParent semaphore before acquiring a mutex lock for itself to update a shared resource, indicating it is finished. The parent is forced to wait until the value of the shared variable shows that all children are finished.

```
$wait(semParent)$
    $// \ critical \ section$
$signal(semParent)$
```

To ensure mutual exclusion for the shared resources (buffer1, buffer2 and counter) the child waited to acquire a mutex lock before entering its critical section to modify the shared resources. Since buffer1 was never modified and only read from this did not require a mutex lock to be obtained before reading.

```
$wait(semMutex)$
    $// \ critical \ section$
$signal(semMutex)$
```

The semaphores required (semMutex and semParent), buffer1, buffer2 and counter were implemented using shared memory. The following POSIX shared memory functions were used to create shared memory, and the corresponding functions to close the shared memory:

$$shm\_link()$$

$$ftruncate()$$

$$mmap()$$

$$shm\_unlink()$$

$$close()$$

$$munmap()$$

Zombie processes were killed with the use of signal(SIGCHILD, SIG_IGN).

2 Thread MSSV

To achieve mutual exclusion in the multi-threaded program, the parent (consumer) must wait for all threads (producer) to complete execution. The parent uses *pthread_lock_mutex()* to lock the mutex then performs a *pthread_cond_wait()* on a condition variable. The condition variable represents how many threads are still executing.

Once the thread completes execution, it acquires a mutex lock to alter the condition variable signaling it has finished its task. The thread also performs a conditional wait in the case where the parent is using the shared variable.

$$pthread\_mutex\_lock(\&mutex)$$

$$pthread\_cond\_wait(\&use, \&mutex)$$

$$pthread\_cond\_signal(\&use);$$

$$pthread\_mutex\_unlock(\&mutex)$$

In the thread MSSV the shared resources are declared as global variables. Threads have access to the variables declared global in the parent. Before a thread could access the shared resources it would first need to acquire a mutex lock. The function *pthread_lock_mutex()* blocks the caller if the mutex is in use by another. It can then alter the shared resources, counter and buffer2, before releasing the mutex lock.

To allocate the memory for buffer1, buffer2, counter and regions *malloc()* was used. The appropriate *free()* calls were used to free the allocated memory. This was done to ensure memory leaks were not present in the operation of the program.

3 Testing

3.1 Method

To test each implementation of MSSV worked as intended multiple input files were used. With these input files, multiple delays were chosen as well. Input files of various 9x9 numbers were used. Input files were used that were valid, contained one error, contained multiple errors. Also tested were smaller grids than 9x9 and empty files.

Testing was performed on the lab machines in various rooms of Building 314, Level 2.

3.2 Errors

There are no known errors in the process MSSV and the thread MSSV. Care has been taken to ensure potential memory leaks are prevented, by using the appropriate measures to free allocated memory. Memory leaks are not present in the testing of each MSSV currently performed.

3.3 Input Files

$$specTest = \begin{matrix} 6 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$testFail = \begin{matrix} 2 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$multiFail = \begin{matrix} 2 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 3 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

3.4 Expected Results

specTest

```
row 1 is valid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
9 out of 9 columns are valid
9 out of 9 sub-grids are valid

There are 27 valid sub-grids, and thus the solution is valid
```

testFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
8 out of 9 columns are valid
8 out of 9 sub-grids are valid

There are 24 valid sub-grids, and thus the solution is invalid
```

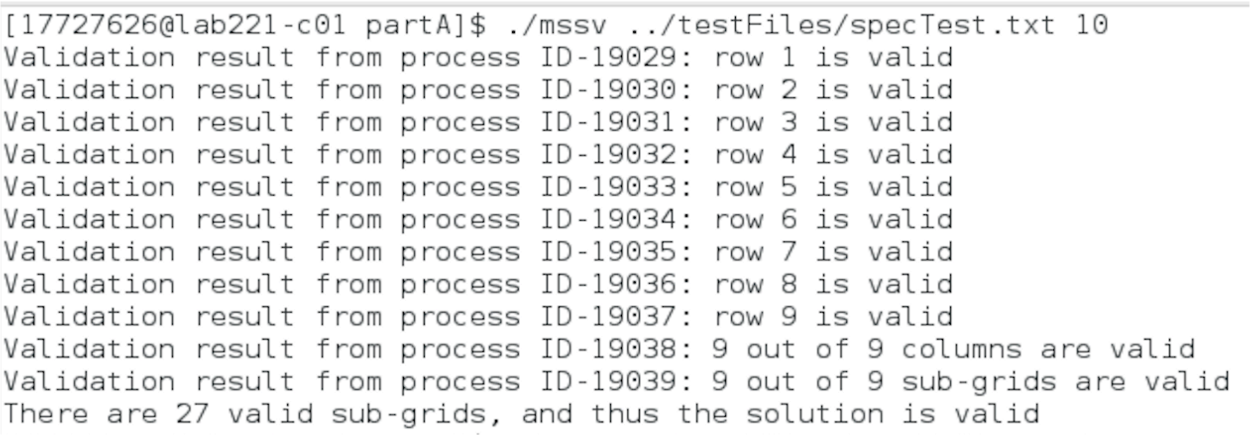
multiFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is invalid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
7 out of 9 columns are valid
7 out of 9 sub-grids are valid

There are 21 valid sub-grids, and thus the solution is invalid
```

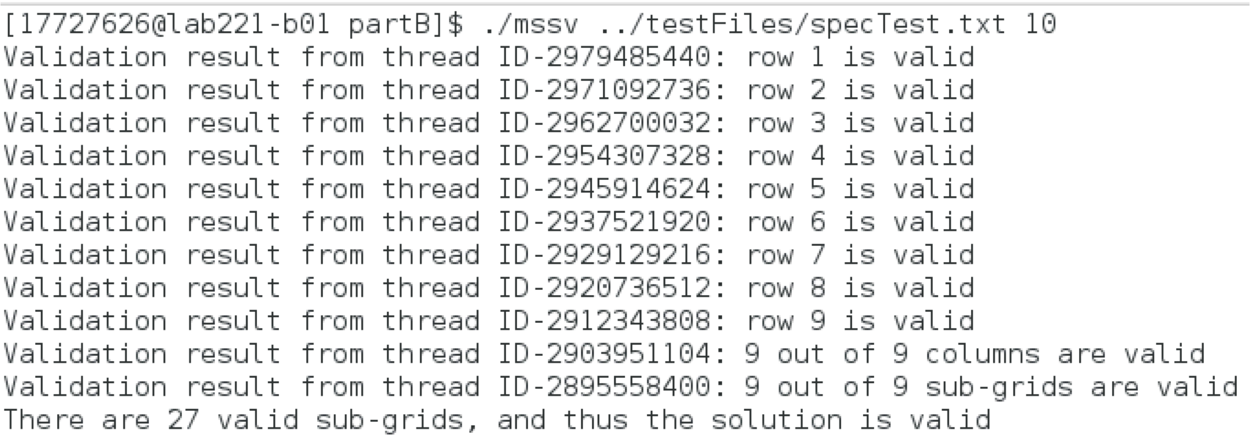
3.5 Actual Results

Figure 1: Processes: specTest



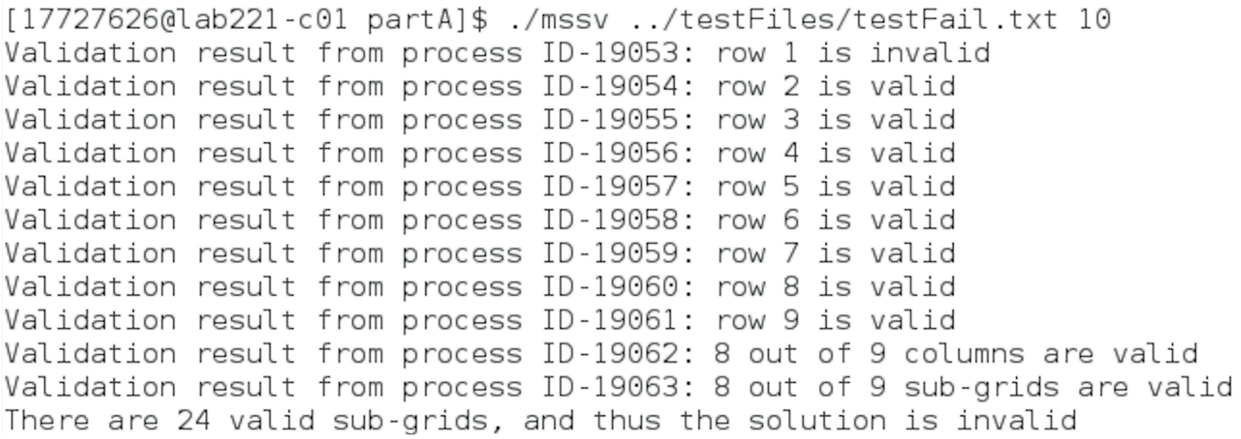
```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/specTest.txt 10
Validation result from process ID-19029: row 1 is valid
Validation result from process ID-19030: row 2 is valid
Validation result from process ID-19031: row 3 is valid
Validation result from process ID-19032: row 4 is valid
Validation result from process ID-19033: row 5 is valid
Validation result from process ID-19034: row 6 is valid
Validation result from process ID-19035: row 7 is valid
Validation result from process ID-19036: row 8 is valid
Validation result from process ID-19037: row 9 is valid
Validation result from process ID-19038: 9 out of 9 columns are valid
Validation result from process ID-19039: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 2: Threads: specTest



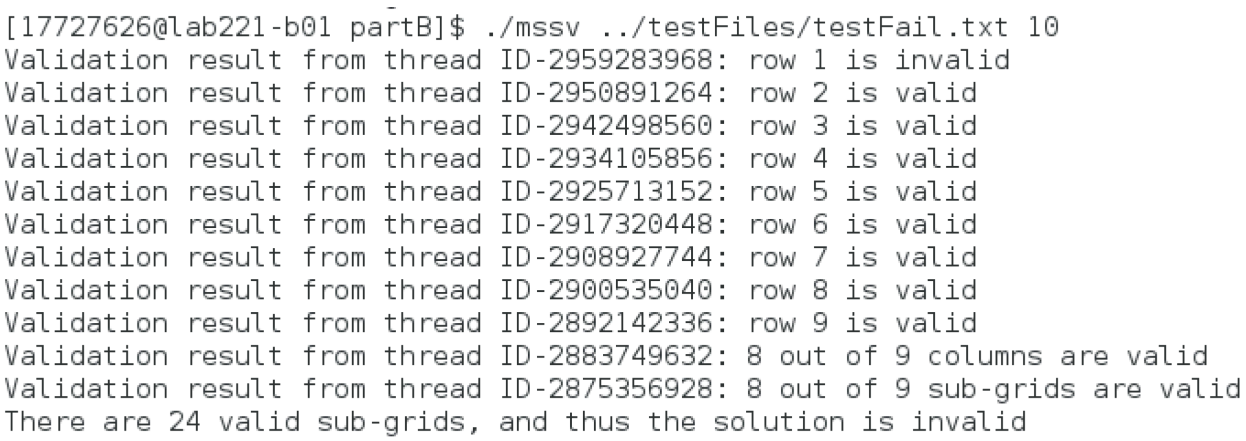
```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/specTest.txt 10
Validation result from thread ID-2979485440: row 1 is valid
Validation result from thread ID-2971092736: row 2 is valid
Validation result from thread ID-2962700032: row 3 is valid
Validation result from thread ID-2954307328: row 4 is valid
Validation result from thread ID-2945914624: row 5 is valid
Validation result from thread ID-2937521920: row 6 is valid
Validation result from thread ID-2929129216: row 7 is valid
Validation result from thread ID-2920736512: row 8 is valid
Validation result from thread ID-2912343808: row 9 is valid
Validation result from thread ID-2903951104: 9 out of 9 columns are valid
Validation result from thread ID-2895558400: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 3: Processes: testFail



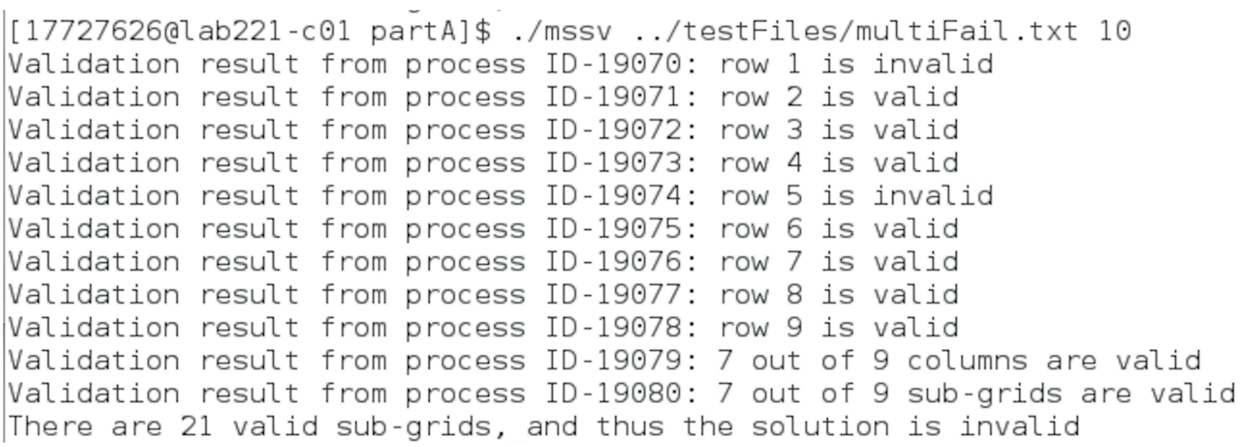
```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/testFail.txt 10
Validation result from process ID-19053: row 1 is invalid
Validation result from process ID-19054: row 2 is valid
Validation result from process ID-19055: row 3 is valid
Validation result from process ID-19056: row 4 is valid
Validation result from process ID-19057: row 5 is valid
Validation result from process ID-19058: row 6 is valid
Validation result from process ID-19059: row 7 is valid
Validation result from process ID-19060: row 8 is valid
Validation result from process ID-19061: row 9 is valid
Validation result from process ID-19062: 8 out of 9 columns are valid
Validation result from process ID-19063: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 4: Threads: testFail



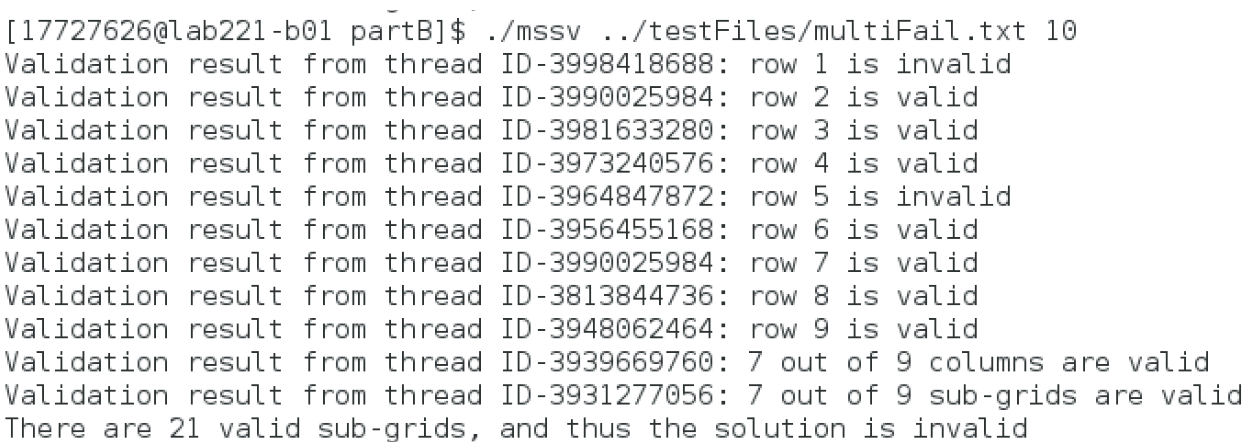
```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/testFail.txt 10
Validation result from thread ID-2959283968: row 1 is invalid
Validation result from thread ID-2950891264: row 2 is valid
Validation result from thread ID-2942498560: row 3 is valid
Validation result from thread ID-2934105856: row 4 is valid
Validation result from thread ID-2925713152: row 5 is valid
Validation result from thread ID-2917320448: row 6 is valid
Validation result from thread ID-2908927744: row 7 is valid
Validation result from thread ID-2900535040: row 8 is valid
Validation result from thread ID-2892142336: row 9 is valid
Validation result from thread ID-2883749632: 8 out of 9 columns are valid
Validation result from thread ID-2875356928: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 5: Processes: multiFail



```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from process ID-19070: row 1 is invalid
Validation result from process ID-19071: row 2 is valid
Validation result from process ID-19072: row 3 is valid
Validation result from process ID-19073: row 4 is valid
Validation result from process ID-19074: row 5 is invalid
Validation result from process ID-19075: row 6 is valid
Validation result from process ID-19076: row 7 is valid
Validation result from process ID-19077: row 8 is valid
Validation result from process ID-19078: row 9 is valid
Validation result from process ID-19079: 7 out of 9 columns are valid
Validation result from process ID-19080: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

Figure 6: Threads: multiFail



```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from thread ID-3998418688: row 1 is invalid
Validation result from thread ID-3990025984: row 2 is valid
Validation result from thread ID-3981633280: row 3 is valid
Validation result from thread ID-3973240576: row 4 is valid
Validation result from thread ID-3964847872: row 5 is invalid
Validation result from thread ID-3956455168: row 6 is valid
Validation result from thread ID-3990025984: row 7 is valid
Validation result from thread ID-3813844736: row 8 is valid
Validation result from thread ID-3948062464: row 9 is valid
Validation result from thread ID-3939669760: 7 out of 9 columns are valid
Validation result from thread ID-3931277056: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

4 README

4.1 Purpose

The program validates an input file containing a sudoku solution. There are two versions. One utilising processes and the other utilising threads.

4.2 Running the Program

To compile the C files into an executable format.

```
  make
```

To run the program there are two options.

Option 1:

```
  make run
```

This will only let you run with the preset parameters and they will need to be altered in the Makefile to test other *input files* and *delays*

```
INPUT: ../testFiles/specTest.txt
DELAY: 10
```

Option 2:

```
./mssv <inputFile> <delay>
```

Between each time the program is run, the following command should be entered and executed. This is to delete the logfile produced from an invalid test file.

```
  make clean
```

4.3 Files

partA

- Makefile
- mssv.c
- mssv.h

partB

- Makefile
- mssv.c
- mssv.h

testFiles

- test files

References

[1] *Interprocess communication using POSIX Shared Memory in Linux.* URL: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux.

[2] *POSIX Semaphores.* URL: https://www.softprayog.in/programming/posix-semaphores.

[3] *POSIX Threads Programming in C.* URL: https://www.softprayog.in/programming/posix-threads-programming-in-c.

[4] *POSIX Threads Synchronization in C.* URL: https://www.softprayog.in/programming/posix-threads-synchronization-in-c.

5 Appendices

5.1 Processes: mssv.c

```c
#include "mssv.h"

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    int maxDelay = atoi(argv[2]);

    // Variables
    int i, pid, processNum, numbers[] = {0,0,0,0,0,0,0,0,0};
    Region* region;
    sem_t semMutex, semParent, *semaphores;

    // File Descriptors
    int buff1FD, buff2FD, counterFD, semFD, regionFD, resFD;

    // Shared memory pointers
    int *buff2Ptr, *countPtr, *resourceCount, (*buff1Ptr)[NINE][NINE];

    // Generate random maxDelay
    srand((unsigned) time(NULL));

    maxDelay = rand() % maxDelay;

    // Create shared memory
    initMemory( &buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD);

    // Map shared memory to pointers
    mapMemory(&buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD,
                    &buff1Ptr, &buff2Ptr, &countPtr, &semaphores, &region,
                        &resourceCount);

    // Initialise semaphores
    if ((sem_init(&semMutex, 1, 1)== 1) ||(sem_init(&semParent, 1, 1) == 1))
    {
        fprintf(stderr, "Could not initialise semaphores\n");
        exit(1);
    }

    semaphores[0] = semMutex;
    semaphores[1] = semParent;

    // Initialise parameters
    *countPtr = 0;
    pid = -1;
    processNum = 0;

    // Read input file
    readFile(inputFile, NINE, NINE, buff1Ptr);

    // Parent aquires lock of resourceCount
    sem_wait(&(semaphores[1])); // Lock child

    *resourceCount = 0;

    // Create child processes for
    while( processNum < NUMPROCESSES && pid != 0 )
    {
        signal(SIGCHLD, SIG_IGN); // Kill zombie processNum-1
        pid = fork();

        // Allow the parent to increment shared variable count
        if ( pid > 0)
        {
            *resourceCount = *resourceCount + 1;
        }
        processNum++;
    }

    if( pid == 0) // Child process
    {
        childManager(region, semaphores, buff1Ptr, buff2Ptr, countPtr,
                            resourceCount,  processNum, numbers, maxDelay );
    }
    else if ( pid > 0) // Parent process
    {
        parentManager(region, semaphores, countPtr, resourceCount);

        // Clean up shared memory
        cleanMemory(&buff1Ptr, &buff2Ptr, &countPtr, &semaphores,
                        &region, &resourceCount, buff1FD, buff2FD, counterFD,
                            semFD, regionFD, resFD);
    }
    else // Unsuccessful child process creation attempt
    {
        fprintf(stderr, "Unable to create child processes. Please run \"killall mssv\"\n");
    }
}

/****************************************************************************/

/**
 * Read the contents of the input file passed as a command line argument
 * @param inputFile File to be read
 * @param rows      Number of rows in matrix
 * @param cols      Number of columns in matrix
 * @param buffer    Matrix to store contents of input file
 */
void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols])
{
    FILE* inStrm;
    int i, j;

    inStrm = fopen(inputFile, "r"); // Open file for reading

    if (inStrm == NULL) // Check file opened correctly
    {
        perror("Error opening file for reading\n");
        exit(1);
    }

    // Store contents of file in 2D array
    for( i = 0; i < rows; i++ )
    {
        for ( j = 0; j < cols; j++ )
        {
            fscanf( inStrm, "%d", &(*(buffer))[i][j] );
        }
    }

    fclose(inStrm); // Close file
}

/**
 * Write the invalid regions to log file
 * @param region Sub region
 * @param format String to be written
 */
```

```c
void writeFile(Region* region, char* format)
{
    char* filename = "logfile";
    FILE* outFile;
    int val;

    outFile = fopen(filename, "a"); // Open file for appending
    if (outFile == NULL) // Check file opened correctly
    {
        perror("Error opening file for writing\n");
        exit(1);
    }

    fprintf(outFile, "process ID-%d: %s",region->pid, format);

    fclose(outFile); // Close file
}

/**
 * Set each index to zero
 * @param numbers Array to be reset
 */
void resetArray(int numbers[])
{
    for (int i = 0; i < NINE; i++)
    {
        numbers[i] = 0;
    }
}

/**
 * Check if the contents of the array has any value other than one
 * @param  numbers Array to be checked
 * @return         Status of array being valid or not
 */
int checkValid(int numbers[])
{
    for (int j = 0; j < NINE; j++)
    {
        if ( numbers[j] != 1)
        {
            return FALSE;
        }
    }

    return TRUE;
}

/**
 * Handles the routine for the parent process. Outputs the result to the screen
 * @param region        Array containing each region struct
 * @param semaphores    Array of all semaphores
 * @param countPtr      Pointer to shared memory counter
 * @param resourceCount Status of number of child processes executing
 */
void parentManager(Region *region, sem_t *semaphores, int* countPtr,
                        int* resourceCount)
{
    char *type, *message;
    sem_post(&(semaphores[1])); // Unlock child
    int done = FALSE;
    int position;

    while( !done ) // Wait for all children to finish executing
    {
        //printf("Parent Waiting for Children\n");
        sem_wait(&(semaphores[1])); // Lock child
        if ( *resourceCount == 0)
        {
            done = TRUE;
        }
        sem_post(&(semaphores[1])); // Unlock child

    }

    for(int ii = 0; ii < NUMPROCESSES; ii++)
    {
        sem_wait(&(semaphores[0])); //Lock mutex
        if (region[ii].type == ROW)
        {
            position = region[ii].positionX;
            type = "invalid";

            if (region[ii].valid == TRUE)
            {
                type = "valid";
            }
            printf("Validation result from process ID-%d: row %d is %s\n",
                                    region[ii].pid, position, type);
        }
        else if (region[ii].type == COL)
        {
            type = "column";
            position = region[ii].positionX;
            printf("Validation result from process ID-%d: %d out of 9 columns are valid\n"
                        ,region[ii].pid, region[ii].positionX);
        }
        else
        {
            type = "sub-grid";
            position = region[ii].positionX;

            printf("Validation result from process ID-%d: %d out of 9 sub-grids are valid\n"
                        ,region[ii].pid, region[ii].positionX);

        }

        sem_post(&(semaphores[0])); //Unlock mutex
    }


  if (*countPtr == 27)
  {
       message = "valid";
  }
  else
  {
      message = "invalid";
  }

  printf("There are %d valid sub-grids, and thus the solution is %s\n", *countPtr, message);
}


/**
 * Routine for child processes. Check the validity of sub region.
 * @param region        Sub-region struct for each process
 * @param semaphores    Array of semaphores
 * @param buff1Ptr      Pointer to buffer1 in shared memory
 * @param buff2Ptr      Pointer to buffer2 in shared memory
 * @param countPtr      Pointer to counter in shared memory
 * @param resourceCount Pointer to resourceCount in shared memory
 * @param processNum    Child process number
 * @param numbers       Array of numbers to check validity of sub region
 * @param maxDelay      Delay for each process
 */
```

```c
void childManager(Region *region, sem_t *semaphores, int (*buff1Ptr)[NINE][NINE],
                    int *buff2Ptr, int* countPtr, int* resourceCount,
                        int processNum, int *numbers, int maxDelay )
{
    char format[500];
    int numValid, comma = 0;

    if( processNum <= 9) // Check a row in buffer1
    {
        for (int i = 0; i < NINE; i++)
        {
            // Update numbers array
            numbers[((*buff1Ptr)[processNum-1][i])-1]++;
        }

        sleep(maxDelay); // Sleep
        sem_wait(&(semaphores[0])); //Lock mutex

        // Update region struct
        region[processNum-1].type = ROW;
        region[processNum-1].positionX = processNum;
        region[processNum-1].pid = getpid();
        region[processNum-1].valid = checkValid(numbers);

        numValid = 0;
        if (region[processNum-1].valid == TRUE)
        {
            numValid = 1;
        }
        else // Write to log file
        {
            sprintf(format, "row %d is invalid\n", processNum);
            writeFile(&(region[processNum-1]), format);
        }

        buff2Ptr[processNum-1] = numValid; // Update buffer2

        *countPtr = *countPtr + numValid; // Update counter

        sem_post(&(semaphores[0])); // Unlock child

    }
    else if(processNum == 10) // Check all columns
    {
        sprintf(format, "column ");

        int validCol = 0;
        for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
        {
            for(int ii = 0; ii < NINE; ii++) // Iterate through each row
            {
                numbers[(*buff1Ptr)[ii][nn]-1]++; // Update numbers array
            }

            if ( checkValid( numbers) == TRUE )
            {
              validCol++;
            }
            else
            {
                if (comma == 0)
                {
                    comma = 1;
                    sprintf(format + strlen(format), "%d", nn+1);
                }
                else
                {
                    sprintf(format + strlen(format), ", %d ", nn+1);
                }
            }

            resetArray(numbers);
        }

        sleep(maxDelay);
        if (validCol == 8)
        {
            sprintf(format + strlen(format), " is invalid\n");
        }
        else
        {
            sprintf(format + strlen(format), "are invalid\n");
        }
        sem_wait(&(semaphores[0])); //Lock mutex

        // Update region struct
        region[processNum-1].type = COL;
        region[processNum-1].positionX = validCol;
        region[processNum-1].pid = getpid();
        if(validCol != 9)
        {
            writeFile(&(region[processNum-1]), format);
        }

        numValid = region[processNum-1].positionX;

        buff2Ptr[processNum-1] = validCol; // Update buffer2

        *countPtr = *countPtr + validCol; // Update counter

        sem_post(&(semaphores[0])); // Unlock mutex
    }
    else if( processNum == 11) // Check sub-grids
    {
        sprintf(format, "sub-grid ");

        int validSub = 0;

        // Iterate through each of the 9 3x3 sub-grid
        for ( int jj = 0; jj < 3; jj++)
        {
            for (int kk = 0; kk < 3; kk++)
            {
                for (int ll = jj*3; ll < jj*3+3; ll++)
                {
                    for (int mm = kk*3; mm < kk*3+3; mm++)
                    {
                        // Update numbers array
                    numbers[(*buff1Ptr)[ll][mm]-1]++;
                  }
              }

              if ( checkValid(numbers) == TRUE )
              {
                    validSub++;
              }
              else // Update string for log file
              {
                  if (comma == 0)
                  {
                      comma = 1;
                      sprintf(format+strlen(format), "[%d..%d, %d..%d]",
                                    jj*3+1, jj*3+3, kk*3+1, kk*3+3);
                  }
                  else
                  {
                      sprintf(format+strlen(format), ", [%d..%d, %d..%d] ",
                                    jj*3+1, jj*3+3, kk*3+1, kk*3+3);
                  }
              }
              resetArray(numbers);
            }

        }

        sleep(maxDelay);
        if (validSub == 8)
        {
            sprintf(format+strlen(format), " is invalid\n");
        }
        else
        {
            sprintf(format+strlen(format), "are invalid\n");
        }
        sem_wait(&(semaphores[0])); //Lock mutex

        // Update region struct
        region[processNum-1].type = SUB_REGION;
        region[processNum-1].positionX = validSub;
        region[processNum-1].pid = getpid();

        if(validSub != 9) // Write to log file
        {
            writeFile(&(region[processNum-1]), format);
        }

        buff2Ptr[processNum-1] = validSub; // Update buffer 2

        *countPtr = *countPtr + validSub; // Update counter

        sem_post(&(semaphores[0])); // Unlock mutex
    }

    // Child signals it is finished by incremented resourceCount
    sem_wait(&(semaphores[1])); // Lock child
        *resourceCount = *resourceCount - 1;
    sem_post(&(semaphores[1])); // Unlock child

}

/**
 * Initalise shared memory constructs
 * @param buff1FD    File descriptor for buffer1
 * @param buff2FD    File descriptor for buffer2
 * @param counterFD File descriptor for counter
 * @param semFD      File descriptor for semaphores
 * @param regionFD   File descriptor for regions
 * @param resFD      File descriptor for resourceCount
 */
void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD)
{

    // Create shared memory
    *buff1FD = shm_open("buffer1", O_CREAT | O_RDWR, 0666);
    *buff2FD = shm_open("buffer2", O_CREAT | O_RDWR, 0666);
    *counterFD = shm_open("counter", O_CREAT | O_RDWR, 0666);
    *semFD = shm_open("semaphores", O_CREAT | O_RDWR, 0666);
    *regionFD = shm_open("region", O_CREAT | O_RDWR, 0666);
    *resFD = shm_open("resources", O_CREAT | O_RDWR, 0666);

    // Check shared memory was created correctly
    if ( *buff1FD == -1 || *buff2FD == -1 || *counterFD == -1 || *semFD == -1 ||
            *regionFD == -1 || *resFD == -1 )
    {
        fprintf( stderr, "Error creating shared memory blocks\n" );
        exit(1);
    }

    // Set size of shared memory constructs
    if ( ftruncate(*buff1FD, sizeof(int) * NINE * NINE) == -1 )
    {
        fprintf( stderr, "Error setting size of buffer1" );
        exit(1);
    }

    if ( ftruncate(*buff2FD, sizeof(int) * NUMPROCESSES) == -1 )
    {
        fprintf( stderr, "Error setting size of buffer2" );
        exit(1);
    }

    if ( ftruncate(*counterFD, sizeof(int)) == -1 )
    {
        fprintf( stderr, "Error setting size of counter" );
        exit(1);
    }

    if ( ftruncate(*semFD, sizeof(sem_t) * 2 ) == -1 )
    {
        fprintf( stderr, "Error setting size of semaphores" );
        exit(1);
    }

    if ( ftruncate(*regionFD, sizeof(Region)*NUMPROCESSES) == -1 )
    {
        fprintf( stderr, "Error setting size of regions" );
        exit(1);
    }

    if ( ftruncate(*resFD, sizeof(int)) == -1 )
    {
        fprintf( stderr, "Error setting size of resourceCount" );
        exit(1);
    }
}

/**
 * Map shared memory to addresses
 * @param buff1FD       File descriptor for buffer1
 * @param buff2FD       File descriptor for buffer2
 * @param counterFD     File descriptor for counter
 * @param semFD         File descriptor for semaphores
 * @param regionFD      File descriptor for regions
 * @param resFD         File descriptor for resourceCount
 * @param buff1Ptr      Pointer to buffer1 in shared memory
 * @param buff2Ptr      Pointer to buffer2 in shared memory
 * @param countPtr      Pointer to counter in shared memory
 * @param semaphores    Array of semaphores
 * @param region        Array of region structs
 * @param resourceCount Pointer to resourceCount in shared memory
 */
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE], int (**buff2Ptr),
                      int** countPtr, sem_t** semaphores, Region** region, int**
    resourceCount)
{
    // Memory mapping
    *buff2Ptr = (int*) mmap(NULL, sizeof(int)*NINE*NINE,
                PROT_READ | PROT_WRITE, MAP_SHARED, *buff2FD, 0);
    *buff1Ptr = mmap(NULL, sizeof(int)*NUMPROCESSES,
                PROT_READ | PROT_WRITE, MAP_SHARED, *buff1FD, 0);
```

```c
    *countPtr = (int*) mmap(NULL, sizeof(int),
                PROT_READ | PROT_WRITE, MAP_SHARED, *counterFD, 0);
    *semaphores = mmap(NULL, sizeof(sem_t) * 2,
                PROT_READ | PROT_WRITE, MAP_SHARED, *semFD, 0);
    *region = mmap(NULL, sizeof(Region)*NUMPROCESSES,
                PROT_READ | PROT_WRITE, MAP_SHARED, *regionFD, 0);
    *resourceCount = mmap(NULL, sizeof(int),
                PROT_READ | PROT_WRITE, MAP_SHARED, *resFD, 0);
}

/**
 * Validate command line parameters
 * @param argc number of parameters
 * @param argv command line parameters
 */
void validateUse(int argc, char* argv[])
{
    // Ensure correct number of command line parameters
    if (argc != 3)
    {
        printf("Ensure there are the correct number of parameters\n");
        exit(1);
    }

    // Ensure maxDelay is positive
    if ( atoi(argv[2]) < 0)
    {
        printf("The maxDelay must be non-negative\n");
        exit(1);
    }
}

/**
 * Close and destroy, semaphores and shared memory constructs
 * @param buff1Ptr      Pointer to buffer1 in shared memory
 * @param buff2Ptr      Pointer to buffer2 in shared memory
 * @param countPtr      Pointer to counter in shared memory
 * @param semaphores    Array of semaphores
 * @param region        Array of region structs
 * @param resourceCount Pointer to resourceCount in shared memory
 * @param buff1FD       File descriptor for buffer1
 * @param buff2FD       File descriptor for buffer2
 * @param counterFD     File descriptor for counter
 * @param semFD         File descriptor for semaphores
 * @param regionFD      File descriptor for regions
 * @param resFD         File descriptor for resourceCount
 */
void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
                         sem_t **semaphores, Region **region,
                            int** resourceCount, int buff1FD, int buff2FD,
                                int counterFD, int semFD, int regionFD,
                                    int resFD )
{
    // Close semaphores
    sem_close(&((*semaphores)[0]));
    sem_close(&((*semaphores)[1]));

    // Destroy semaphores
    sem_destroy(&((*semaphores)[0]));
    sem_destroy(&((*semaphores)[1]));

    // Clean up shared memory
    shm_unlink("buffer1");
    shm_unlink("buffer2");
    shm_unlink("counter");
    shm_unlink("semaphores");
    shm_unlink("region");
    shm_unlink("resources");

    // Close file descriptors
    close(buff1FD);
    close(buff2FD);
    close(counterFD);
    close(semFD);
    close(regionFD);
    close(resFD);

    // Unmap memory
    munmap(*buff1Ptr, sizeof(int)*NINE*NINE);
    munmap(*buff2Ptr, sizeof(int)*NUMPROCESSES);
    munmap(*countPtr, sizeof(int));
    munmap(*semaphores, sizeof(sem_t)*2);
    munmap(*region, sizeof(Region)*NUMPROCESSES);
    munmap(*resourceCount, sizeof(int));

    // Unlink shared memory constructs
    shm_unlink("buffer1");
    shm_unlink("buffer2");
    shm_unlink("counter");
    shm_unlink("semaphores");
    shm_unlink("region");
    shm_unlink("resources");

}
```

5.2 Processes: mssv.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>

#define NINE 9
#define SUB 3
#define NUMPROCESSES 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_REGION} Region_Type;

typedef struct
{
    Region_Type type;
    int positionX;
    pid_t pid;
    int valid;
} Region;


void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols]);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(Region *region, sem_t *semaphores, int* countPtr,
                        int* resourceCount );
void childManager(Region *region, sem_t *semaphores,
                    int (*buff1Ptr)[NINE][NINE], int *buff2Ptr, int* countPtr,
                        int* resourceCount, int processNum, int *numbers,
                            int maxDelay );
void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
                    sem_t **semaphores, Region **region, int** resourceCount,
                         int buff1FD, int buff2FD, int counterFD, int semFD,
                            int regionFD, int resFD);
```

5.3 Processes: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 1

# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
	$(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
	$(CC) -c mssv.c $(CFLAGS)

clean:
	rm -f $(EXEC1) $(OBJ1) logfile

run:
	./$(EXEC1) $(INPUT) $(DELAY)
```

5.4 Threads: mssv.c

```
#include ”mssv.h”

pthread_mutex_t mutex; // Mutex
pthread_cond_t use;  // condition for if the global variable is in use

int **buff1, *buff2, *counter, maxDelay, inUse;
Region *regions;

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    maxDelay = atoi(argv[2]);

    // Variables
    pthread_t threads[11];

    // Generate random maxDelay
    srand((unsigned) time(NULL));
    maxDelay = rand() % maxDelay;

    // Allocate  memory
    initMemory( &buff1, &buff2, &counter, &regions);

    *counter = 0;
    // Read input file
    readFile(inputFile, NINE, NINE, &buff1);

    // Initialise mutex and condition
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&use, NULL);
    inUse = 0;

    // Create threads
    for(int i = 0; i < NUMTHREADS; i++)
    {
       if ( i < NINE) // Initialise region struct for row threads
       {
           regions[i].type = ROW;
       }
       else if( i == NINE) // Initialise region struct for columns thread
       {
           regions[i].type = COL;
       }
       else // Initialise region struct for sub-grids thread
       {
           regions[i].type = SUB_GRID;
       }

       regions[i].position = i;
       resetArray(regions[i].numbers);
       // Create thread
       pthread_create(&(threads[i]), NULL, childManager, &(regions[i]));
       inUse++;
    }

    parentManager(); // Parent logic

    cleanMemory(); // Clean up malloc'd memory

}

/****************************************************************************/
```

```

/**
 * Initalise memory constructs
 * @param buff1   buffer1 2D array
 * @param buff2   buffer2 1D array
 * @param counter counter variable
 * @param regions Region struct 1D array
 */
void initMemory(int*** buff1, int** buff2, int** counter, Region** regions)
{
    // Initialise
    *buff1 = (int**) malloc(sizeof(int*)* NINE);
    for (int i = 0; i < NINE; i++)
    {
        (*buff1)[i] = (int*) malloc(sizeof(int)* NINE);
    }
    *buff2 = (int*) malloc(sizeof(int)* NUMTHREADS);
    *counter = (int*) malloc(sizeof(int));
    *regions = (Region*) malloc(sizeof(Region)* NUMTHREADS);
}


/**
 * Free malloc'd memory and destroy mutex and conditions
 */
void cleanMemory()
{
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&use);
    for (int i = 0; i < NINE; i++)
    {
        free(buff1[i]);
    }
    free(buff1);
    free(buff2);
    free(counter);
    free(regions);
}


/**
 * Read the contents of the input file passed as a command line argument
 * @param inputFile File to be read
 * @param rows      Number of rows in matrix
 * @param cols      Number of columns in matrix
 * @param buffer    Matrix to store contents of input file
 */
void readFile(char* inputFile, int rows, int cols, int***buffer )
{
    FILE* inStrm;
    int i, j;

    inStrm = fopen(inputFile, ”r”); // Open file for reading

    if (inStrm == NULL) // Check file opened correctly
    {
        perror(”Error opening file for reading\n”);
        exit(1);
    }

    // Store contents of file in 2D array
    for( i = 0; i < rows; i++ )
    {
        for ( j = 0; j < cols; j++ )
        {
            fscanf( inStrm, ”%d”, &(*(buffer))[i][j] );
        }
    }
    fclose(inStrm); // Close file
}

/**
 * Write the invalid regions to log file
 * @param region Sub region
 * @param format String to be written
 */
void writeFile(Region* region, char* format)
{
    char* filename = "logfile";
    FILE* outFile;
    int val;

    outFile = fopen(filename, "a"); // Open file for appending
    if (outFile == NULL) // Check file opened correctly
    {
        perror("Error opening file for writing\n");
        exit(1);
    }

    fprintf(outFile, "thread ID-%d: %s",region->tid, format);

    fclose(outFile); // Close file
}

/**
 * Set each index to zero
 * @param numbers Array to be reset
 */
void resetArray(int numbers[])
{
    for (int i = 0; i < NINE; i++)
    {
        numbers[i] = 0;
    }
}

/**
 * Check if the contents of the array has any value other than one
 * @param  numbers Array to be checked
 * @return         Status of array being valid or not
 */
int checkValid(int numbers[])
{
    for (int j = 0; j < NINE; j++)
    {
        if ( numbers[j] != 1)
        {
            return FALSE;
        }
    }

    return TRUE;
}

/**
 * Handles the routine for the parent. Outputs the result to the screen
 * @param threads ID of child threads
 */
void parentManager()
{
    char *type, *message;
    int done = FALSE;
    int position;

    pthread_mutex_lock(&mutex); // Lock mutex
    while ( inUse > 0 ) // Wait while children are executing
    {
        pthread_cond_wait(&use, &mutex);
    }

    pthread_cond_signal(&use);
    pthread_mutex_unlock(&mutex); // Unlock mutex

    for(int ii = 0; ii < NUMTHREADS; ii++)
    {
        pthread_mutex_lock(&mutex); // Lock mutex

        if (regions[ii].type == ROW)
        {
            type = "row";
            position = regions[ii].position;
            if ( regions[ii].valid == TRUE)
            {
                printf("Validation result from thread ID-%u: %s %d is valid\n",
                                        regions[ii].tid,type, position+1);
            }
            else
            {
                printf("Validation result from thread ID-%u: %s %d is invalid\n",
                          regions[ii].tid, type, position+1);
            }
        }
        else if (regions[ii].type == COL)
        {
            type = "column";
            printf("Validation result from thread ID-%u: %d out of 9 columns are valid\n"
                    ,regions[ii].tid, regions[ii].count);
        }
        else
        {
            type = "sub-grid";
            printf("Validation result from thread ID-%u: %d out of 9 sub-grids are valid\n"
                    ,regions[ii].tid, regions[ii].count);
        }

        pthread_mutex_unlock(&mutex);
    }


  if (*counter == 27)
  {
       message = "valid";
  }
  else
  {
      message = "invalid";
  }

  printf("There are %d valid sub-grids, and thus the solution is %s\n", *counter, message);

}

/**
 * Routine for child threads. Check the validity of sub region.
 * @param args Void pointer to Region struct for the child
 */
void* childManager(void* args )
{
    char format[500];
    int numValid;
    Region* region = ((Region*)(args));
    int threadNum = region->position;
    int comma = 0;

    if( region->type == ROW ) // Check row in buffer1
    {

        // Check rows
        for (int i = 0; i < NINE; i++)
        {
            // Update numbers array
            region->numbers[((buff1)[threadNum][i])-1]++;
        }

        sleep(maxDelay); // Sleep
        pthread_mutex_lock(&mutex); // Lock mutex

        // Update region struct
        region->tid = pthread_self();
        region->valid = checkValid(region->numbers);

        // Update buffer2
        numValid = 0;
        if (region->valid == TRUE)
        {
            numValid = 1;
            region->count = numValid;
        }
        else // Write to log file
        {
            region->count = numValid;
            sprintf(format, "row %d is invalid\n", threadNum+1);
            writeFile((region), format);
        }

        buff2[threadNum] = numValid; // Update buffer2

        *counter = *counter + numValid; // Update counter

        pthread_mutex_unlock(&mutex); // Unlock mutex

    }
    else if( region->type == COL ) // Check all columns
    {
        sprintf(format, "column ");
        int validCol = 0;
        for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
        {
            for(int ii = 0; ii < NINE; ii++) // Iterate through each row
            {
                // Update numbers array
                region->numbers[((buff1)[ii][nn])-1]++;
            }

            // Check if the column is valid
            if ( checkValid( region->numbers) == TRUE )
            {
              validCol++;
            }
            else
            {
                if (comma == 0)
                {
                    comma = 1;
                    sprintf(format + strlen(format), "%d", nn+1);
                }
                else
                {
                    sprintf(format + strlen(format), ", %d ", nn+1);
                }
            }
            resetArray(region->numbers);
        }

        sleep(maxDelay);
        if (validCol == 8)
        {
            sprintf(format + strlen(format), " is invalid\n");
        }
        else
        {
            sprintf(format + strlen(format), "are invalid\n");
        }
        pthread_mutex_lock(&mutex); // Lock mutex

        // Update region struct
        region->count = validCol;
        region->tid = pthread_self();
        if(validCol != 9)
        {
            writeFile((region), format);
        }

        numValid = region->count;

        buff2[threadNum] = validCol; // Update buffer2

        *counter = *counter + validCol; // Update counter

        pthread_mutex_unlock(&mutex); // Unlock mutex
    }
    else if( region->type == SUB_GRID ) // Check all sub-grids
    {
        sprintf(format, "sub-grid ");

        int validSub = 0;

        // Iterate through each of the 9 3x3 sub-grid
        for ( int jj = 0; jj < 3; jj++)
        {
            for (int kk = 0; kk < 3; kk++)
            {
                for (int ll = jj*3; ll < jj*3+3; ll++)
                {
                    for (int mm = kk*3; mm < kk*3+3; mm++)
                    {
                        // Update numbers array
                        region->numbers[((buff1)[ll][mm])-1]++;
                    }
                }

                if ( checkValid(region->numbers) == TRUE )
                {
                    validSub++;
                }
                else // Update string for log file
                {
                    if (comma == 0)
                    {
                        comma = 1;
                        sprintf(format+strlen(format), "[%d..%d, %d..%d]",
                                jj*3+1, jj*3+3, kk*3+1, kk*3+3);
                    }
                    else
                    {
                        sprintf(format+strlen(format), ", [%d..%d, %d..%d]",
                                jj*3+1, jj*3+3, kk*3+1, kk*3+3);
                    }
                }
                resetArray(region->numbers);
            }

        }

        sleep(maxDelay);
        if( validSub == 8)
        {
            sprintf(format+strlen(format), " is invalid\n");
        }
        else
        {
            sprintf(format+strlen(format), " are invalid\n");
        }
        pthread_mutex_lock(&mutex); // Lock mutex

        // Update region struct
        region->count= validSub;
        region->tid = pthread_self();

        if(validSub != 9)
        {
            writeFile((region), format);
        }

        buff2[threadNum] = validSub; // Update buffer2

        *counter = *counter + validSub; // Update counter

        pthread_mutex_unlock(&mutex); // Unlock mutex

    }

    // Child signals it is finished by incremented resourceCount
    pthread_mutex_lock(&mutex); // Lock mutex

    while( inUse == 0)
    {
        pthread_cond_wait(&use, &mutex);
    }
    inUse--; // Decrease count of child processes running
    if (inUse == 0)
    {
        pthread_cond_signal(&use);
    }
    pthread_mutex_unlock(&mutex); // Unlock mutex
    pthread_detach(pthread_self()); // Release resources

}

/**
 * Validate command line parameters
 * @param argc number of parameters
 * @param argv command line parameters
 */
void validateUse(int argc, char* argv[])
{
    // Ensure correct number of command line parameters
    if (argc != 3)
    {
        printf("Ensure there are the correct number of parameters\n");
        exit(1);
    }

    // Ensure maxDelay is positive
    if ( atoi(argv[2]) < 0)
    {
        printf("The maxDelay must be non-negative\n");
        exit(1);
    }
}
```

5.5 Threads: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#define NINE 9
#define SUB 3
#define NUMTHREADS 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_GRID} Region_Type;

typedef struct
{
    Region_Type type;
    int position;
    pthread_t tid;
    int count;
    int valid;
    int numbers[NINE];

} Region;


void readFile(char* inputFile, int rows, int cols, int***buffer);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(void);
void* childManager(void* args );
void initMemory( int*** buff1, int** buff2, int** counter, Region** regions);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(void);
```

5.6 Threads: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -g -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 10


# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
	$(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
	$(CC) -c mssv.c $(CFLAGS)

#fileIO.o : fileIO.c fileIO.h
#	$(CC) -c fileIO.c $(CFLAGS)

clean:
	rm -f $(EXEC1) $(OBJ1) logfile

run:
	./$(EXEC1) $(INPUT) $(DELAY)
```